OpenACC Assignment 2

Joseba Hernández Bravo and Jorge Vicente Puig

November 2021

Universitat Autònoma de Barcelona

Laplace Algorithm

Question 1

The compiler returns the following information about `laplace_error` function:

```
laplace_error:
     16, Loop not vectorized: data dependency
         Loop unrolled 2 times
     18, stencil inlined, size=3 (inline) file lap2.c (7)
```

which indicates that it hasn't been able to know if the loops were independent and then it could not vectorized the loops. Then, when using `perf stat` we obtain the following measurements,

Table 1: Measures on `lap2.c` CPU execution

Elapsed time	IPC	Machine instructions
8,571269820	1,44	41.131.725.610

which even thought we have an acceptable IPC, we know that the performance have to be improve at least by vectorizing the loop.

Question 2

When using the keyword *restrict* on function definition, i.e.

```
    float laplace_error ( float * restrict old, float * restrict new, int n, int m )
```

we are avoiding pointer aliasing and then the compiler is able to vectorized the loops which is shown by the message,

```
        laplace_error:
     16, Generated vector simd code for the loop containing reductions
     18, stencil inlined, size=3 (inline) file lap2.c (7)
```

which indicates firstly that the loop has been vectorized and secondly recognises that there is a reduction clause to take care on the loop *(the one that compute the error)*.

Then, we can compare the measurements of this execution with the previous one,

Table 2: Vectorized and not vectorized measurements of `lap2.c`

Version	Elapsed time	IPC	Machine instructions
Not vectorized	8,571269820	1,44	41.131.725.610
Vectorized	2,906166020	1,11	10.577.233.335

We can observe that, as expected, when the compiler vectorizes the `laplace_error` function the number of machine instructions is reduced around by 4 times, which implies in a reduction of elapsed time by around 3 times.

Question 3

For making a GPU parallelization with OpenACC the following directives have been added to the `lap2.c` code,

```
#pragma acc data copy(A[:n*m]) copyin(Anew[:n*m])
```

this directive help the compiler to understand two important facts. Firstly, implicitly we are indicating that the matrices have to be copied to the GPU memory from the host and we want back the matrix `A` to the host (even thought it would be better if we also return `Anew` to prevent problems from pointer swapping procedure). Secondly, by indicating that we only have to do memory operations at the beginning and ending of the numerical method we are avoiding data copies during it.

The other directive have to be add on the `laplace_error` function,

```
#pragma acc kernels loop independent collapse(2)  present(old,new)
```

which indicates firstly that we are securing the compiler that the loop can be collapsed and then paralellized, secondly we are indicating that the matrices, that are given to the function as pointers, are already in another data region.

Using this code version on GPU execution we obtain the following measurements:

Table 3: Different number of iterations on `lap2.c`, GPU-accelerated

Nº iterations	**Elapsed time**
100	0,229144672
1000	0,515849383
10000	3,487851911

We can see that it has been done a great improvement by using the GPU. In fact, if we compare the result with those of the the previous tasks (that were also executed with 100 iterations) we will notice that the execution time in this case is reduced from $8,51$s or 2.906s to $0.22s$. This, corresponds to a speedup of :

- $8,51/0.22 = 38.6818\times$ compared to the 1st task.
- $2,906/0.22 = 13.209\times$ compared to the 2nd task.

Moreover, is important to notice that when increasing the number of iterations, the elapsed time do not increase proportionally. Due to the big bandwidth provided by the GPU we are able to obtain better results when increasing the problem size.

We are also able to check using the provided `profGPU.txt` script that there is not an evident bottleneck on our program. Data movements required few time (less than one second) and the 70% of elapsed time is used for the `laplace_error` whereas 29% is for the reduction clause, which seems reasonably.

Question 4

When using the `-acc=multicore` option on the compiler we obtain the following measures,

Table 4: Different versions of `lap2.c` with 100 iterations

Version	**Elapsed time**	**IPC**	**Machine instructions**
Single-thread CPU	2,906166020	1,11	10.577.233.335
Multi-thread CPU	1,823906372	0,29*4	6.375.642.165
GPU-accelerated	0,229144672	XXX	XXX

Note that the **perf** command provides information from the CPU-s point of view. Thus, the only metric that is useful in the the case of the GPU-accelerated version is the **elapsed time**. We will substitute the remaining useful values by **XXX**.

We notice that using the multicore code we have obtained better results than for the sequential program: elapsed time is reduced by 1,6 times and the number of machine instructions by 1,65. Additionally, we have obtained an IPC value of 0,29. Since we have used 4 cores in the execution, by multiplying the number of cores by the IPC of each one, we obtain a total IPC of 1,16, which is slightly higher than then one obtained with the Single-thread execution.

In order to get a deeper understanding, we measure the executions with different number of iterations,

Table 5: Different number of iterations on `lap2.c`, Multi-thread CPU

Nº iterations	**Elapsed time**	**IPC**	**Machine instructions**
100	1,823906372	0,29*4	6.375.642.165
1000	17,622131705	0,28*4	61.748.364.073
10000	151,263073266	0,28*4	536.076.240.210

Firstly, we notice that the program do not have a good scale when increasing the problem size as happened with the GPU version. The increase of elapsed time and machine instructions is almost proportional to the increase on number of iterations. Secondly, by looking at the IPC is confirmed that there is not any improvement when increasing the problem size, even, the IPC slightly decrease.

Linear-Equations

Question 1

When invoking the compiler with the option `-Minfo=accel` for the case of a single-thread CPU execution, it provides the following information:

```
102, Accelerator restriction: size of the GPU copy of A is unknown
     Complex loop carried dependence of A-> prevents parallelization
     Loop carried dependence of xnew-> prevents parallelization
     Loop carried backward dependence of xnew-> prevents vectorization
     Complex loop carried dependence of b->,xold-> prevents parallelization
104, Loop is parallelizable
113, Loop is parallelizable
```

The compiler is telling us that the loops in lines 104 and 113 are parallelizable. However, it also tell us that the loop in in line 103, which correspond to the main iteration of the numerical method is not able to know if the data is dependent.

In this case, as we are running the program with a single thread, the pragma directives will be omitted. However, it is useful for the case of both multi-thread and GPU execution. After executing the program using the **perf stat** instrument, we obtain the following results:

Table 6: Different number of iterations on `jsolvec.cpp`, Single-thread CPU

Nº iterations	**Elapsed time**	**IPC**	**Machine instructions**
1000	4,685225001	2,83	44.386.581.736
5000	23,042049811	2,84	220.600.283.459
10000	42,141311810	2,84	403.464.482.439

Since it is difficult to know if the program is performing well with $\sim 40G$ of machine instructions, it is more useful to rely on the IPC value. Therefore, considering that we have obtained a value near 3 and, taking into account that the maximum of this value is 4, we can conclude that our program makes good performance. Moreover, is important to observe that elapsed time and machine instructions increase proportionally with the increase on the number of iterations.

Question 2

In order to parallelize the program with the proper directives we have to think about two strategies. First, we would like to parallelize the outer loop by using,

```
#pragma acc kernels loop independent
```

Then, we can do a data movement, before the while loop, from the host to the device as follow:

```
#pragma acc data copyin(A[:nsize*nsize], b[:nsize])
```

While using this directives, the compiler tell us the following message:

```
 98, Generating copyin(b[:nsize],A[:nsize*nsize]) [if not already present]
105, Loop is parallelizable
     Generating Tesla code
    105, #pragma acc loop gang /* blockIdx.x */
    107, #pragma acc loop vector(128) /* threadIdx.x */
         Generating implicit reduction(+:rsum)
105, Generating implicit copyin(xold[:nsize]) [if not already present]
     Generating implicit copyout(xnew[:nsize]) [if not already present]
107, Loop is parallelizable
116, Loop is parallelizable
     Generating Tesla code
    116, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         Generating implicit reduction(+:residual)
116, Generating implicit copyin(xold[:nsize],xnew[:nsize]) [if not already present]
     Generating implicit copy(residual) [if not already present]
```

which shows how was able to notice the 2 reductions operations, how was able to parallelize the loops and the data operations that will be done during execution.

Then, with this directives we achieve the following results:

Table 7: Different number of iterations on `jsolvec.cpp`, accelerated GPU

Nº iterations	**Elapsed time**
1000	0,361010013
5000	0,873274016
10000	1,420994628

As we can see, the results obtained are much better than those obtained when running on a single-thread CPU. In fact, for 1000, 5000 and 10000 iterations, the improvements have been multiplied by ~ 13, ~ 26 and ~ 29 respectively.